arXiv:1109.3989v2 [cs.PL] 11 Oct 2011

The SeaLion has Landed: An IDE for Answer-Set Programming—Preliminary Report*

Johannes Oetsch, Jörg Pührer, and Hans Tompits

Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. We report about the current state and designated features of the tool SeaLion, aimed to serve as an integrated development environment (IDE) for answer-set programming (ASP). A main goal of SeaLion is to provide a user-friendly environment for supporting a developer to write, evaluate, debug, and test answer-set programs. To this end, new support techniques have to be developed that suit the requirements of the answer-set semantics and meet the constraints of practical applicability. In this respect, SeaLion benefits from the research results of a project on methods and methodologies for answer-set program development in whose context SeaLion is realised. Currently, the tool provides source-code editors for the languages of Gringo and DLV that offer syntax highlighting, syntax checking, and a visual program outline. Further implemented features are support for external solvers and visualisation as well as visual editing of answer sets. SeaLion comes as a plugin of the popular Eclipse platform and provides itself interfaces for future extensions of the IDE.

1 Introduction

Answer-set programming (ASP) is a well-known and fully declarative problem-solving paradigm based on the idea that solutions to computational problems are represented in terms of logic programs such that the models of the latter, referred to as the *answer sets*, provide the solutions of a problem instance.[1] In recent years, the expressibility of languages supported by answer-set solvers increased significantly [3]. As well, ASP solvers have become much more efficient, e.g., the solver Clasp proved to be competitive with state-of-the-art SAT solvers [4].

Despite these improvements in solver technology, a lack of suitable *engineering tools* for developing programs is still a handicap for ASP towards gaining widespread popularity as a problem-solving paradigm. This issue is clearly recognised in the ASP community and work to fill this gap has started recently, addressing issues like debugging, testing, and the modularity of programs [5–13]. Additionally, in order to facilitate tool support as known for other programming languages, attempts to provide *integrated development environments* (IDEs) have been put forth. Work in this direction includes the systems APE [14], ASPIDE [15], and iGROM [16].

Following this endeavour, in this paper, we describe the current status and designated features of a further IDE, SeaLion, developed as part of an ongoing research project on methods and methodologies for developing answer-set programs [17].

SeaLion is designed as an Eclipse plugin, providing useful and intuitive features for ASP. Besides experts, the target audience for SeaLion are software developers new to ASP, yet who are familiar with support tools as used in procedural and object-oriented programming. Our goal is to fully support the languages of the current state-of-the-art solvers Clasp (in conjunction with Gringo) [3, 18] and DLV [19], which distinguishes SeaLion from the other IDEs mentioned above which support only a single solver. Indeed, APE [14], which is also an Eclipse plugin, supports only the language of Lparse [20] that is a subset of the language of Gringo, whilst ASPIDE [15], a recently developed standalone IDE, offers support only for DLV programs. Although iGROM provides basic functionality for the languages of both Lparse and DLV [16], it currently does not support the latest version of DLV or the full syntax of Gringo.

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698.

[1] For an overview about ASP, we refer the reader to a survey article by Gelfond and Leone [1] or the textbook by Baral [2].

At present, SeaLion is in an alpha version that already implements important core functionality. In particular, the languages of DLV and Gringo are supported to a large extent. The individual parsers translate programs and answer sets to data structures that are part of a rich and flexible framework for internally representing program elements. Based on these structures, the editor provides syntax highlighting, syntax checks, error reporting, error highlighting, and automatic generation of a program outline. There is functionality to manage external tools such as answer-set solvers and to define arbitrary pipes between them (as needed when using separate grounders and solvers). Moreover, in order to run an answer-set solver on the created programs, launch configurations can be created in which the user can choose input files, a solver configuration, command line arguments for the solver, as well as output-processing strategies. Answer sets resulting from a launch can either be parsed and stored in a view for interpretations, or the solver output can be displayed unmodified in Eclipse's built-in console view.

Another key feature of SeaLion is the capability for the *visualisation* and *visual editing* of interpretations. This follows ideas from the visualisation tools ASPVIZ [21] and IDPDraw [22], where a visualisation program Π_V (itself being an answer-set program) is joined with an interpretation I that shall be visualised. Subsequently, the overall program is evaluated using an answer-set solver, and the visualisation is generated from a resulting answer set. However, the editing feature of SeaLion allows also to graphically manipulate the interpretations under consideration which is not supported by ASPVIZ and IDPDraw.

The visualisation functionality of SeaLion is itself represented as an Eclipse plugin, called Kara.[2] In this paper, however, we describe only the basic functionality of Kara; a full description is given in a companion paper [23].

2 Architecture and Implementation Principles

We assume familiarity with the basic concepts of answer-set programming (ASP) (for a thorough introduction to the subject, cf. Baral [2]). In brief, an answer-set program consists of rules of the form

$$a_1 \vee \cdots \vee a_l :- a_{l+1}, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n,$$

where $n \geq m \geq l \geq 0$, "not" denotes *default negation*, and all a_i are first-order literals (i.e., atoms possibly preceded by the *strong negation* symbol $\neg$). For a rule r as above, the expression left to the symbol "$:-$" is the *head* of r and the expression to the right of "$:-$" is the *body* of r. If $n = l = 1$, r is a *fact*; if r contains no disjunction, r is *normal*; and if $l = 0$ and $n > 0$, r is a *constraint*. For facts, the symbol "$:-$" is usually omitted. The *grounding* of a program P relative to its Herbrand universe is defined as usual. An *interpretation* I is a finite and consistent set of ground literals, where consistency means that $\{a, \neg a\} \not\subseteq I$, for any atom a. I is an *answer set* of a program P if it is a minimal model of the grounding of the *reduct* of P relative to I (see Baral [2] for details).

A key aspect in the design of SeaLion is extensibility. That is, on the one hand, we want to have enough flexibility to handle further ASP languages such that previous features can deal with them with no or little adaption. On the other hand, we want to provide a powerful API framework that can be used by future features. To this end, we defined a hierarchy of classes and interfaces that represent *program elements*, i.e., fragments of ASP languages. This is done in a way such that we can use common interfaces and base classes for representing similar program elements of different ASP languages. For instance, we have different classes for representing literals of the Gringo language and literals of the DLV language in order to be able to handle subtle differences. For example, in Gringo, a literal can have several other literals as conditions, e.g., `redEdge(X,Y):edge(X,Y):red(X):red(Y)`. Intuitively, during grounding, this literal is replaced by the list of all literals `redEdge(n1,n2)`, where `edge(n1,n2)`, `red(n1)`, and `red(n2)` can be derived during grounding. As DLV is unaware of conditions, an object of class DLVStandardLiteral has no support for them, whereas a GringoStandardLiteral object keeps a list of condition literals. Substantial differences in other language features, like aggregates,

[2] The name derives, with all due respect, from "Kara Zor-El", the native Kryptonian name of *Supergirl*, given that Kryptonians have visual superpowers on Earth.

optimisation, and filtering support, are also reflected by different classes for Gringo and DLV, respectively. However, whenever possible, these classes are derived from a common base class or share common interfaces. Therefore, plugins can, for example, use a general interface for aggregate literals to refer to aggregates of both languages. Hence, current and future feature implementations can make use of high-level interfaces and stay independent of the concrete ASP language to a large extent.

Also, within the SeaLion implementation, the aim is to have independent modules for different features, in form of Eclipse plugins, that ensure a well-structured code. Currently, there are the following plugins: (i) the main plugin, (ii) a plugin that adapts the ANTLR parsing framework [24] to our needs, (iii) two solver plugins, one for Gringo/Clasp and one for DLV, and (iv) the Kara plugin for answer-set visualisation and visual editing. Moreover, it is a key aim to smoothly integrate SeaLion in the Eclipse platform and to make use of functionality the latter provides wherever suitable. The motivation is to exploit the rich platform as well as to ensure compatibility with upcoming versions of Eclipse.

The decision to build on Eclipse, rather than writing a stand-alone application from scratch, has many benefits. For one, we profit from software reuse as we can make use of the general GUI of Eclipse and just have to adapt existing functionality to our needs. Examples include the text editor framework, source-code annotations, problem reporting and quick fixes, project management, the undo-redo mechanism, the console view, the navigation framework (Outline, Project Explorer), and launch configurations. Moreover, much functionality of Eclipse can be used without any adaptions, e.g., workspace management, the possibility to define working sets, i.e., grouping arbitrary files and resources together, software versioning and revision control (e.g., based on SVN or CVS), and task management. Another clear benefit is the popularity of Eclipse among software developers, as it is a widely used standard tool for developing Java applications. Arguably, people who are familiar with Eclipse and basic ASP skills will easily adapt to SeaLion. Finally, choosing Eclipse for an IDE for ASP offers a chance for integration of development tools for hybrid languages, i.e., combinations of ASP and procedural languages. For instance, Gringo supports the use of functions written in the LUA scripting language [25]. As there is a LUA plugin for Eclipse available, one can at least use that in parallel with SeaLion, however there is also potential for a tighter integration of the two plugins.

The sources of SeaLion are available for download from

`http://sourceforge.net/projects/mmdasp/.`

An Eclipse update site will be made available as soon as SeaLion reaches beta status.

3 Current Features

In this section, we describe the features that are already operational in SeaLion, including technical details on the implementation.

3.1 Source-Code Editor

The central element in SeaLion is the *source-code editor* for logic programs. For now, it comes in two variations, one for DLV and one for Gringo. A screenshot of a Gringo source file in SeaLion's editor is given in Fig. 1. By default, files with names ending in ".lp", ".lparse", ".gr", or ".gringo" are opened in the Gringo editor, whereas files with extensions ".dlv" or ".dl" are opened in the DLV editor. Nevertheless, any file can be opened in either editor if required.

The editors provide *syntax highlighting*, which is computed in two phases. Initially, a fast syntactic check provides initial colouring and styling for comments and common tokens like dots concluding rules and the rule implication symbol. While editing the source code, after a few moments of user inactivity, the source code is parsed and data structures representing the program are computed and stored for various purposes. The second phase of syntax highlighting is already based on this program representation and allows for fine-grained highlighting depending not only on the type of the program element but also on its role. For instance, a literal that is used in the condition of another literal is highlighted in a different way than stand-alone literals.

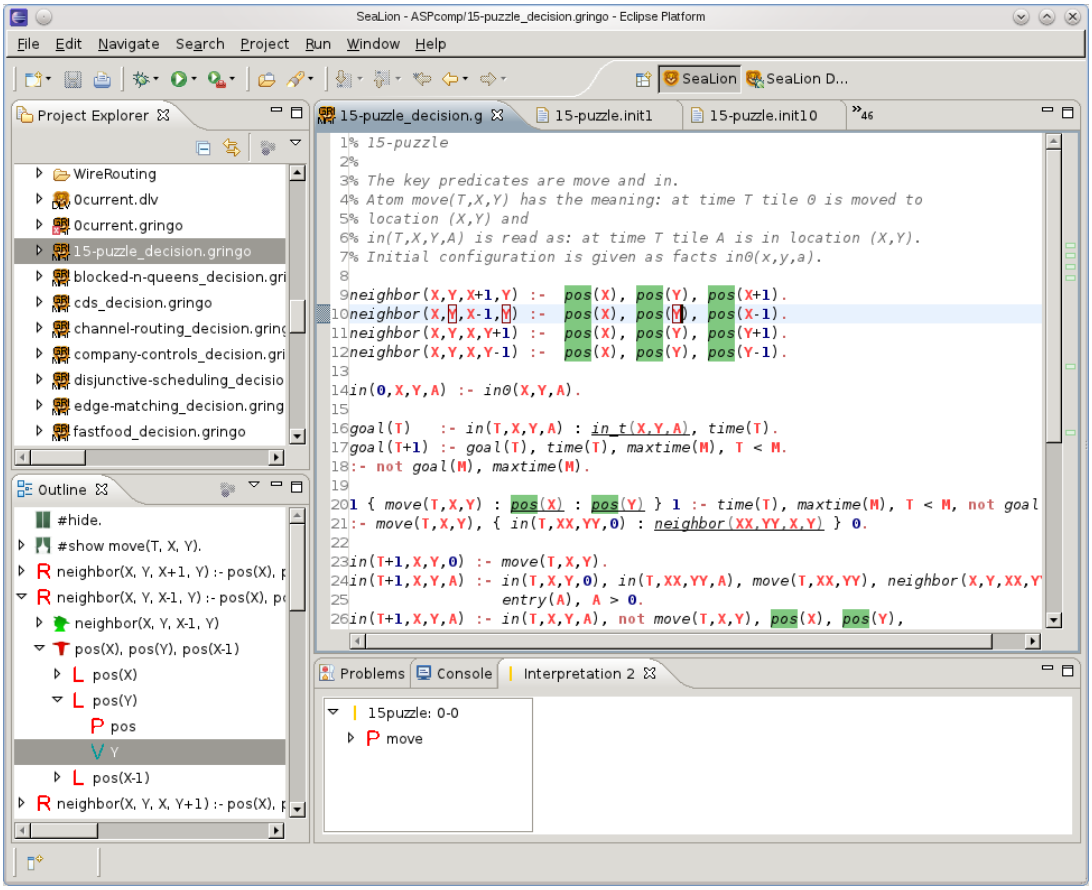

Fig. 1. A screenshot of SeaLion's editor, the program outline, and the interpretation view.

The parsers used are based on the ANTLR framework [24] and are in some respect *more lenient than the respective solver parsers*. For one thing, they are more tolerant towards syntax errors. For instance, in many cases they accept terms of various types (constants, variables, aggregate terms) where a solver requires a particular type, like a variable. The errors will still be noticed, during building the program representation or afterwards, by means of explicit checks. This tolerance allows for more specific warning and error reporting than provided by the solvers. For example, the system can warn the user that he or she used a constant on the left-hand side of an assignment where only a variable is allowed. Another parsing difference is the *handling of comments*. The parser does not throw them away but collects them and associates them to the program elements in their immediate neighbourhood. One benefit is that the information contained in comments can be kept when performing automatic transformations on the program, like rule reorderings or translations to other logic programming dialects. Another advantage is that we can make use of comments for enriching the language with our own *meta-statements* that do not interfere with the solver when running the file. We reserved the token "%!" for initiating meta commands and "%*!" and "*%" for the start and end of block meta commands in the Gringo editor, respectively. Currently, one type of meta command is supported: assigning properties to program elements.

Example 1. In the following source code, a meta statement assigns the name "r1" to the rule it precedes.

```
%! name = r1;
a(X) :- c(X).
```

These names are currently used in a side application of SeaLion for reifying disjunctive non-ground programs as used in a previous debugging approach [10]. Moreover, names assigned to program elements as above can be seen in Eclipse's "Outline View". SeaLion uses this view to give an overview of the edited

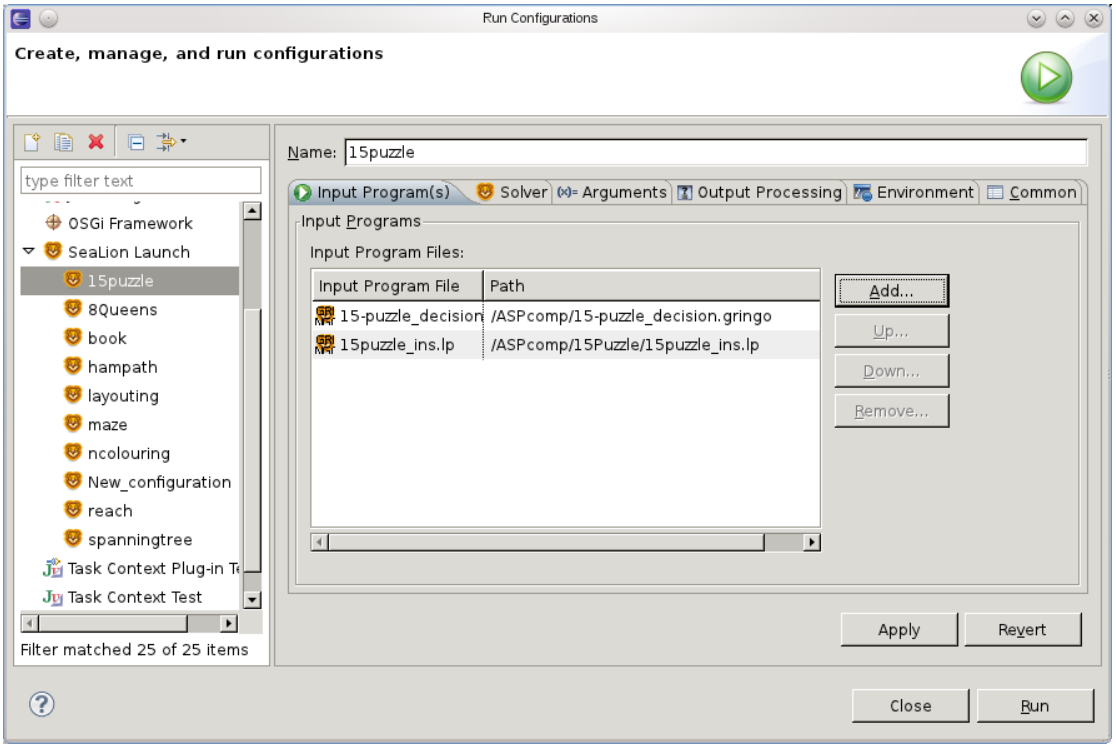

Fig. 2. Selecting two source files for ASP solving in Eclipse's launch configuration dialog.

program in a tree-shaped graphical representation. The rules of the programs are represented by the nodes of depth 1 of this tree. By expanding the ancestor nodes of an individual rule, one can see its elements, i.e., head, body, literals, predicates, terms, etc. Clicking on such an element selects the corresponding program code in the editor, and the programmer can proceed editing there. A similar outline is also available in Eclipse's "Project Explorer", as subtree under the program's source file.

Another feature of the editor is the support for *annotations*. These are means to temporarily highlight parts of the source code. For instance, `SeaLion` annotates occurrences of the program element under the text cursor. If the cursor is positioned over a literal, all literals of the same predicate are highlighted in the text as well as in a bar next to the vertical scrollbar that indicates the positions of all occurrences in the overall document. Likewise, when a constant or a variable in a rule is on the cursor position, their occurrences are detected within the whole source code or within the rule, respectively.

Another application of annotations is *problem reporting*. Syntax errors and warnings are displayed in two ways. First, as annotations in the source code, they are marked with a zig-zag styled underline. Second, they are displayed in Eclipse's "Problem View" that collects various kinds of problems and allows for directly jumping to the problematic source code region upon mouse click.

3.2 Support for External Tools

In order to interact with solvers and grounders from `SeaLion`, we implemented a mechanism for handling external tools. One can define *external tool configurations* that specify the path to an executable as well as default command-line parameters. Arbitrary command-line tools are supported; however, there are special configuration types for some programs such as `Gringo`, `Clasp`, and `DLV`. For these, it is planned to have a specialised GUI that allows for a more convenient modification of command-line parameters. In addition to external command-line tools, one can also define tool configurations that represent pipes between external tools. This is needed when grounding and solving are provided by separate executables. For instance, one can define two separate tool configurations for `Gringo` and `Clasp` and define a piped tool configuration

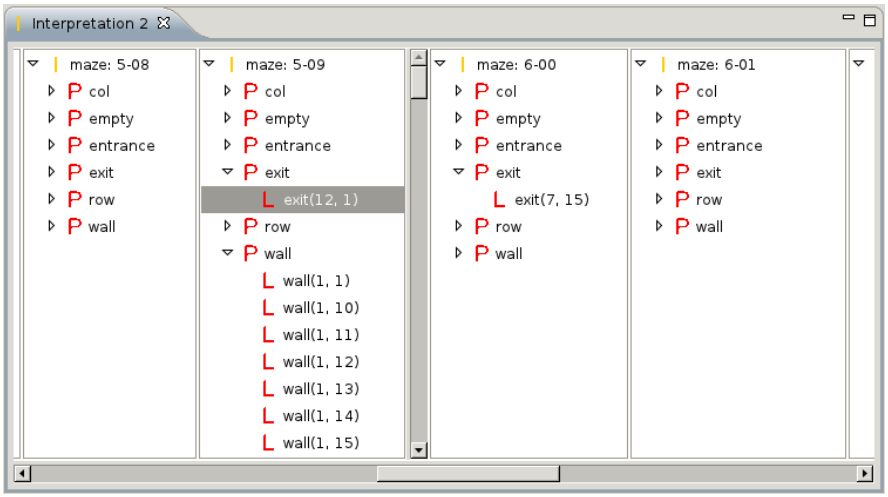

Fig. 3. `SeaLion`'s interpretation view.

for using the two tools in a pipe. Pipes of arbitrary length are supported such that arbitrary pre- and post-processing can be done when needed.

For executing answer-set solvers, we make use of Eclipse's "launch configuration framework". In our setting, a launch configuration defines which programs should be executed using which solver. Figure 2 shows the the page of the launch configuration editor on which input files for a solver invocation can be selected.

Besides using the standard command-line parameters from the tool configurations, also customised parameters can be set for the individual program launches.

3.3 Interpretation View

The programmer can define how the output of an ASP solver run should be treated. One option is to print the solver output as it is for Eclipse's "console view". The other option is to parse the resulting answer sets and store them in `SeaLion`'s *interpretation view* that is depicted in Fig. 3. Here, interpretations are visualised as expandable trees of depth 3. The root node is the interpretation (marked by a yellow "I"), and its children are the predicates (marked by a red "P") appearing in the interpretation. Finally, each of these predicates is the parent node of the literals over the predicate that are contained in the interpretation (marked by a red "L"). Compared to a standard textual representation, this way of visualising answer sets provides a well-arranged overview of the individual interpretations. We find it also more appealing than a tabular representation where only entries for a single predicate are visible at once. Moreover, by horizontally arranging trees for different interpretations next to each other, it is easy to compare two or more interpretations.

The interpretation view is not only meant to provide a good visualisation of results, but also serves as a starting point for ASP developing tools that depend on interpretations. One convenient feature is dragging interpretations or individual literals from the interpretation view and dropping them on the source-code editor. When released, these are transformed into facts of the respective ASP language.

3.4 Visualisation and Visual Editing

The plugin `Kara` [23] is a tool for the graphical visualisation and editing of interpretations. It is started from the interpretation view. One can select an interpretation for visualisation by right-clicking it in the view and choose between a *generic visualisation* or a *customised visualisation*. The latter is specified by the user by means of a visualisation answer-set program. The former represents the interpretation as a labelled hypergraph.

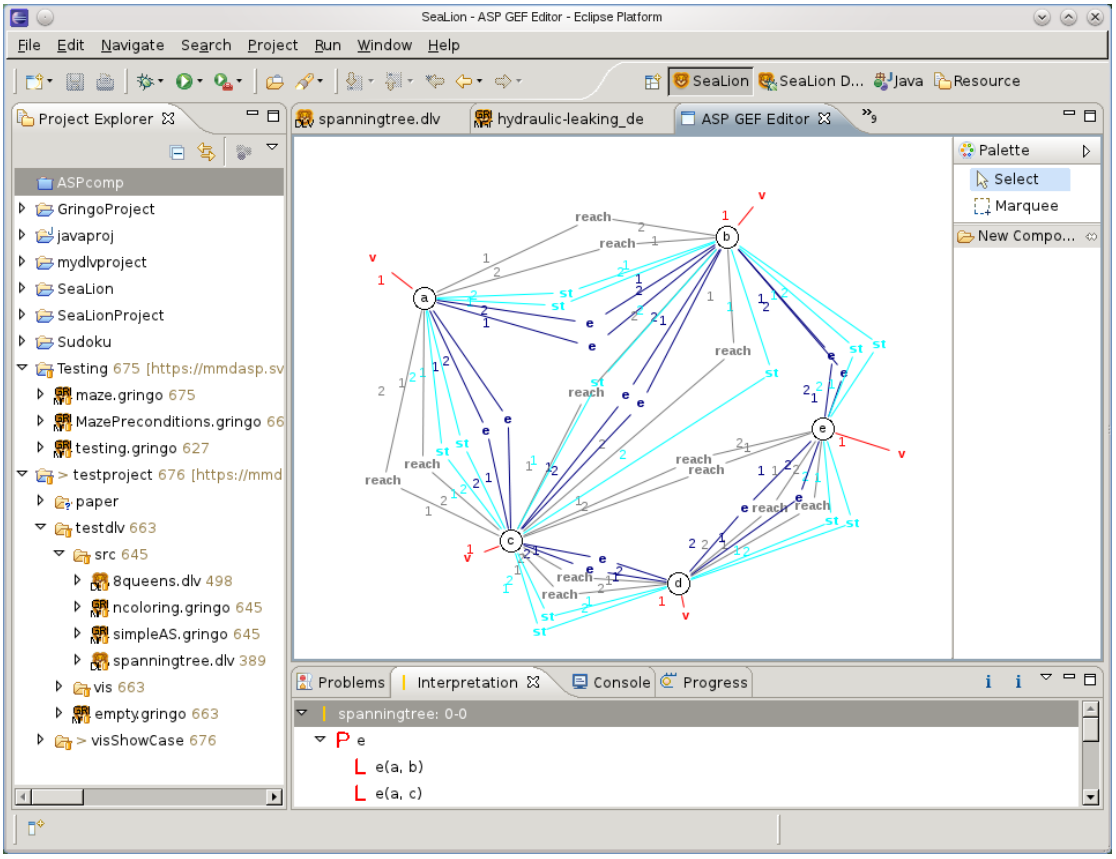

Fig. 4. A screenshot of SeaLion's visual interpretation editor.

In the generic visualisation, the nodes of the hypergraph are the individuals appearing in the interpretation. The edges represent the literals in the interpretation, connecting the individuals appearing in the respective literal. Integer labels on the endings of an edge are used for expressing the argument position of the individual. In order to distinguish between different predicates, each edge has an additional label stating the predicate name. Moreover, edges of the same predicate are of the same colour. An example of a generic visualisation of a spanning tree interpretation is shown in Fig. 4 (the layout of the graph has been manually optimised in the editor).

The customised visualisation feature allows for specifying how the interpretation should be illustrated by means of an answer-set program that uses a powerful pre-defined visualisation vocabulary. The approach follows the ideas of ASPVIZ [21] and IDPDraw [22]: a visualisation program Π_V is joined with the interpretation I to be visualised (technically, I is considered as a set of facts) and evaluated using an answer-set solver. One of the resulting answer sets, I_V, is then interpreted by SeaLion for building the graphical representation of I. The vocabulary allows for using and positioning basic graphical elements such as lines, rectangles, polygons, labels, and images, as well as graphs and grids composed of such elements.

The resulting visual representation of an interpretation is shown in a graphical editor that also allows for manipulating the visualisation in many ways. Properties such as colours, IDs, and labels can be manipulated and graphical elements can be repositioned, deleted, or even created. This is useful for two different purposes. First, for fine-tuning the visualisation before saving it as a scalable vector graphic (SVG) for use outside of SeaLion, using our SVG export functionality. Second, modifying the visualisation can be used to obtain a modified version I' of the visualised interpretation I by abductive reasoning.

In fact, we implemented a feature that allows for abducing an interpretation that would result in the modified visualisation. Modifications in the visual editor are automatically reflected in an adapted version

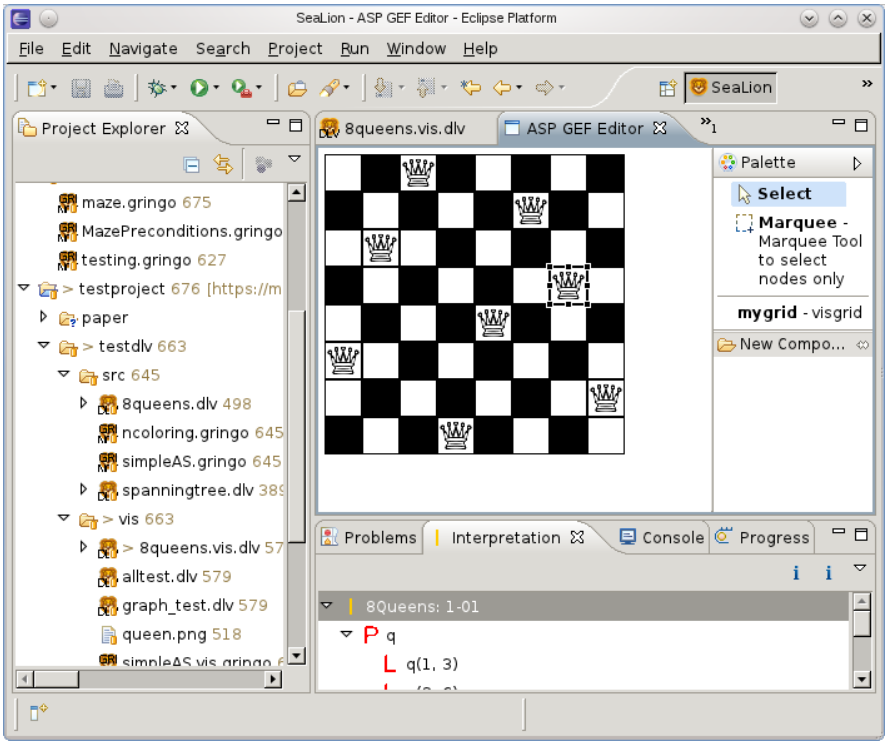

Fig. 5. A customised visualisation of an 8-queens instance.

I'_V of the answer set I_V representing the visualisation. We then use an answer-set program $\lambda(I'_V, \Pi_V)$ that is constructed depending on the modified visualisation answer set I'_V and the visualisation program Π_V for obtaining the modified interpretation I' as a projected answer set of $\lambda(I'_V, \Pi_V)$. For more details, we refer to a companion paper [23]. An example for a customised visualisation for a solution to the 8-queens problem is given in Fig. 5.

4 Projected Features

In the following, we give an overview of further functionality that we plan to incorporate into `SeaLion` in the near future.

One core feature that is already under development is the support for *stepping-based debugging* of answer-set programs as introduced in recent work [26]. Here, we aim for an intuitive and easy-to-handle user interface, which is clearly a challenge to achieve for reasons intrinsic to ASP. In particular, the discrepancy of having non-ground programs but solutions based on their groundings makes the realisation of practical debugging tools for ASP non-trivial.

We want to enrich `SeaLion` with support for *typed predicates*. That is, the user can define the domain for a predicate. For instance consider a predicate `age/2` stating the age of a person. Then, with typing, we can express that for every atom `age(t1,t2)`, the term `t1` represents an element from a set of persons, whereas `t2` represents an integer value. Two types of domain specifications will be supported, namely direct ones, which explicitly state the names of the individuals of the domain, and indirect ones that allow for specifications in terms of the domain of other predicates. We expect multiple benefits from having this kind of information available. First, it is useful as a documentation of the source code. A programmer can clearly specify the intended meaning of a predicate and look it up in the type specifications. Moreover, type violations in the source code of the program can be automatically detected as illustrated by the following example.

Example 2. Assume we want to define a rule deriving atoms with predicate symbol `serves/3`, where `serves(R,D,P)` expresses that restaurant `R` serves dish `D` at price `P`. Furthermore, the two predicates `dishAvailable/2` and `price/3` state which dishes are currently available in which restaurants and the price of a dish in a restaurant, respectively. Assume we have type specifications stating that for `serves(R,D,P)` and `dishAvailable(D,R)`, `R` is of type `restaurant` and `D` of type `dish`. Then, a potential type violation in the rule

```
serves(R,D,P) :- dishAvailable(R,D),price(R,D,P)
```

could be detected, where the programmer mixed up the order of variables in `dishAvailable(R,D)`.

In order to avoid problems like in the above example in the first place, autocompletion functionality could be implemented such that variables and constants of correct types are suggested when writing the arguments of a literal in a rule. Technically, we plan to realise type definitions within program comments, similar to other meta-statements as sketched in Section 3.

We want to combine the typing system with functionality that allows for defining *program signatures*. One application of such signatures is for specifying the predicates and terms used for abducing a modified interpretation I' in our plugin for graphically editing interpretations. Moreover, input and output signatures can be defined for uniform problem encodings, i.e., answer-set programs that expect a set of facts representing a problem instance as input such that its answer sets correspond to the solutions for this instance. Then, such signatures can be used in our planned support for *assertions* that will allow for defining pre- and post-conditions of answer-set programs. Having a full specification for the input of a program, i.e., a typed signature and input constraints in the form of preconditions, one can automatically generate input instances for the program and use them, e.g., for random testing [12]. Also, more advanced testing and verification functionality can be realised, like the automatic generation of valid input (with respect to the pre-conditions) that violates a post-condition.

In order to reduce the amount of time a programmer has to spend for writing type and signature definitions, we want to explore methods for partially extracting them from the source code or from interpretations.

Other projected features include typical amenities of Eclipse editors such as refactoring, autocompletion, pretty-printing, and providing quick-fixes for typical problems in the source code. Moreover, checks for errors and warnings that are not already detected by the parser, for example for detecting unsafe variables, need still to be implemented.

We also want to provide different kinds of program translations in `SeaLion`. To this end, we already implemented a flexible framework for transforming program elements to string representations following different strategies. In particular, we aim at translations between different solver languages at the non-ground level. Here, we first have to investigate strategies when and how transformations of, e.g., aggregates can be applied such that a corresponding overall semantics can be achieved. Other specific program translations that we consider for implementation would be necessary for realising the import and export of rules in the Rule Interchange Format (RIF) [27] which is a W3C recommendation for exchanging rules in the context of the Semantic Web. Notably, a RIF dialect for answer-set programming, called RIF-CASPD, has been proposed [28].

Further convenience improvements regarding the use of external tools in `SeaLion` include the support for setting default solvers for different languages and a specialised GUI for choosing the command-line parameters. For launch configurations, we want to add the possibility to directly write the output of a tool invocation into a file and to allow for exporting the launch configuration as native stand-alone scripts.

Finally, there are many possible ways to enhance the GUI of `SeaLion`. We want to extend the support for drag-and-drop operations such that, e.g., program elements in the outline can be dragged into the editor. Moreover, we plan to realise sorting and filtering features for the outline and interpretation view. Regarding interpretations, we aim for supporting textual editing of interpretations directly in the view, besides visual editing, and a feature for comparing multiple interpretations by highlighting their differences.

5 Related Work

In this section, we give a short overview of existing IDEs for core ASP languages. To begin with, the tool `APE` that has been developed at the University of Bath [14] is also based on Eclipse. It supports

the language of Lparse and provides syntax highlighting, syntax checking, program outline, and launch configuration. Additionally, APE has a feature to display the predicate dependency graph of a program. ASPIDE, a recent IDE for DLV programs [15], is a standalone tool that already offers many features as it builds on previous tools [29–31]. Some functionality we want to incorporate in SeaLion is already supported by ASPIDE, e.g., code completion, refactoring, and quick fixes. Further features of ASPIDE are support for code templates and a visual program editor. We do not aim for comprehensive visual source-code editing in SeaLion but consider the use of program templates that allow for expressing common programming patterns. One disadvantage of ASPIDE is that the tracing component of the IDE [30] is not publicly available. In their current releases, neither APE nor ASPIDE support graphical visualisation or visual editing of answer sets as available in SeaLion. ASPIDE allows for displaying answer sets in a tabular form. This is an improvement compared to the standard textual representation but comes with the drawback that only entries for a single predicate are visible at once. Besides the graphical representation, SeaLion can display interpretations in a dedicated view that gives a good overview of the individual interpretations and allows also to compare different interpretations.

Concerning supported ASP languages, SeaLion is the first IDE to support the language of Gringo, rather than its Lparse subset. Moreover, other proposed IDEs for ASP do only consider the language of either DLV or Lparse, with the exception of iGROM that provides basic syntax highlighting and syntax checking for the languages of both, Lparse and DLV [16]. Note that iGROM has been developed at our department independently from SeaLion as a student project. A speciality of iGROM is the support for the front-end languages for planning and diagnosis of DLV. There also exist proprietary IDEs for ASP related languages with support for object-oriented features, OntoStudio and OntoDLV [32, 33].

Compared to ASPVIZ [21] and IDPDraw [22], our plugin Kara [23] allows not only for visualisation of an interpretation but also for visually editing the graphical representation such that changes are reflected in the visualised interpretation. Moreover, Kara offers support for generic visualisation, automatic layout of graph structures, and special support for grids.

6 Conclusion

In this paper, we presented the current status of SeaLion, an IDE for ASP languages that is currently under development. We discussed general principles that we follow in our implementation and gave an overview of current and planned features. SeaLion is an Eclipse plugin and supports the ASP languages of Gringo and DLV. The most important step in the advancement of the IDE is the integration of an easy-to-use debugging system.

References

1. Gelfond, M., Leone, N.: Logic programming and knowledge representation - The A-Prolog perspective. Artificial Intelligence **138**(1-2) (2002) 3–38
2. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge, England, UK (2003)
3. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver clasp: Progress report. In: Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2009). Volume 5753 of Lecture Notes in Computer Science, Springer (2009) 509–514
4. SAT 2009 competition: http://www.satcompetition.org/2009/
5. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proceedings of the 3rd Workshop on Answer Set Programming: Advances in Theory and Implementation (ASP 2005). Volume 142 of CEUR Workshop Proceedings, Aachen, Germany, CEUR-WS.org (2005)
6. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming **9**(1) (2009) 1–56
7. Syrjänen, T.: Debugging inconsistent answer-set programs. In: Proceedings of the 11th International Workshop on Nonmonotonic Reasoning (NMR 2006). Volume IfI-06-04 of IfI Technical Report Series, Clausthal-Zellerfeld, Germany, Institut für Informatik, Technische Universität Clausthal (2006) 77–83
8. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). Volume 4483 of Lecture Notes in Computer Science, Springer (2007) 31–43

9. Wittocx, J., Vlaeminck, H., Denecker, M.: Debugging for model expansion. In: Proceedings of the 25th International Conference on Logic Programming (ICLP 2009). Volume 5649 of Lecture Notes in Computer Science, Springer (2009) 296–311
10. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: Towards debugging non-ground answer-set programs. Theory and Practice of Logic Programming **10**(4–5) (2010) 513–529
11. Niemelä, I., Janhunen, T., Oetsch, J., Pührer, J., Tompits, H.: On testing answer-set programs. In: Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010). (2010) 951–956
12. Janhunen, T., Niemelä, I., Oetsch, J., Pührer, J., Tompits, H.: Random vs. structure-based testing of answer-set programs: An experimental comparison. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011). Volume 6645 of Lecture Notes in Computer Science, Springer (2011) 242–247
13. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. Journal of Artificial Intelligence Research **35** (August 2009) 813–857
14. Sureshkumar, A., Vos, M.D., Brain, M., Fitch, J.: APE: An AnsProlog* environment. In: Proceedings of the 1st International Workshop on Software Engineering for Answer-Set Programming (SEA 2007). (2007) 71–85
15. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated development environment for answer set programming. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011). Volume 6645 of Lecture Notes in Computer Science, Springer (2011) 317–330
16. iGROM: `http://igrom.sourceforge.net/`
17. Oetsch, J., Pührer, J., Tompits, H.: Methods and methodologies for developing answer-set programs—Project description. In: Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010). Volume 7 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2010)
18. Gebser, M., Schaub, T., Thiele, S.: Gringo : A new grounder for answer set programming. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). (2007) 266–271
19. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
20. Syrjänen, T.: Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
21. Cliffe, O., De Vos, M., Brain, M., Padget, J.A.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: Proceedings of the 24th International Conference on Logic Programming, (ICLP 2008). (2008) 724–728
22. Wittocx, J.: IDPDraw, a tool used for visualizing answer sets. `https://dtai.cs.kuleuven.be/krr/software/visualisation` (2009)
23. Kloimüllner, C., Oetsch, J., Pührer, J., Tompits, H.: `Kara` - A system for visualising and visual editing of interpretations for answer-set programs. In: Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011). (2011)
24. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers. Pragmatic Bookshelf (May 2007)
25. Ierusalimschy, R.: Programming in Lua, Second Edition. Lua.Org (2006)
26. Oetsch, J., Pührer, J., Tompits, H.: Stepping through an answer-set program. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011). Volume 6645 of Lecture Notes in Computer Science, Springer (2011) 134–147
27. Boley, H., Kifer, M., eds.: RIF Framework for Logic Dialects. W3C (2010) W3C Recommendation 22 June 2010.
28. M., K., Heymans, S.: RIF core answer set programming dialect. `http://ruleml.org/rif/RIF-CASPD.html` (2009)
29. Febbraro, O., Reale, K., Ricca, F.: A visual interface for drawing ASP programs. In: Proceedings of the 25th Italian Conference on Computational Logic (CILC 2010). (2010)
30. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proceedings of the 2nd International Workshop on Software Engineering for Answer-Set Programming (SEA 2009), Potsdam, Germany. (2009)
31. Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: spock: A debugging support tool for logic programs under the answer-set semantics. In: Revised Selected Papers of the 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and the 21st Workshop on Logic Programming (WLP 2007). Volume 5437 of Lecture Notes in Computer Science, Springer (2009) 247–252
32. ontoprise GmbH: OntoStudio 3.0. (2010) `http://help.ontoprise.de/`.
33. Ricca, F., Gallucci, L., Schindlauer, R., Dell'armi, T., Grasso, G., Leone, N.: OntoDLV: An ASP-based System for Enterprise Ontologies. Journal of Logic and Computation (2008)